Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Functional Active Objects: Noninterference and Distributed Consensus

Ludovic Henrio and Florian Kammüller

Bericht-Nr. 2009-19
ISSN 1436-9915

Functional Active Objects:
Noninterference and Distributed Consensus

Bericht-Nr. 2009-19

Ludovic Henrio and Florian Kammüller

ISSN 1436-9915

Abstract

In this report, we present recent work on the language of functional active objects ASP_{fun}.

We first introduce briefly the language ASP_{fun}, its syntax and semantics. Then, we present a method for static security checking for our functional distributed active object language. We show how the type system of ASP_{fun} is easily extensible for noninterference: a type system that enables analyzing an ASP_{fun} program statically – prior to execution – detects information flows that contradict a given security policy. To prove this conjecture, we introduce the definition of an indistinguishability relation and prove the noninterference theorem that shows that this indistinguishability relation is a bisimulation on ASP_{fun} executions.

In a second part, we investigate the question of distributed consensus in ASP_{fun}. We implement Paxos, a distributed consensus algorithm due to Lamport, in ASP_{fun}. This implementation illustrates how functional active objects behave when stateful operations occur.

Contents

Chapter 1

$\mathbf{ASP_{fun}}$ Semantics

In this section, we briefly recall the definition of the language ASP_{fun} [8] a functional active object language. The language and a (classical) type system are formalized in Isabelle/HOL; type safety is proved, i.e. preservation and progress implying deadlock-freedom [8]. We present here only the syntax and semantics of ASP_{fun}. In the following section only, we introduce the noninterference type system that is an extension of the classical one.

1.1 Syntax

$$
\begin{array}{rll}
s, t ::= & \underline{x} & \text{variable} \\
| & \underline{[l_j = \varsigma(x_j, y_j)t_j]^{j \in 1..n}} & (\forall j,\, x_j \neq y_j) \text{ object definition} \\
| & \underline{s.l_i(t)} & (i \in 1..n) \text{ method call} \\
| & \underline{s.l_i := \varsigma(x, y)t} & (i \in 1..n, x \neq y) \text{ update} \\
| & \underline{\mathit{Active}(s)} & \text{Active object creation} \\
| & \alpha & \text{active object reference} \\
| & f_i & \text{future reference}
\end{array}
$$

1.2 Semantics

A *configuration* is a set of activities. Each activity possesses a single active object, which is a ς-calculus term. Activating an object, $\mathit{Active}(s)$, means creating a new activity with the object s to be activated becoming an *active object*.[1] It is immutable. A request sent to an activity is an invocation to the active object; it is processed by the activity. The set of requests processed by an activity is called *request queue* by similarity with the active object model but, here, as the calculus is functional, requests can be treated in an unordered fashion.

$$C ::= \alpha_i[(f_j \mapsto s_j)^{j \in I_i}\,,\, t_i]^{i \in 1..p} \quad \text{where } \{I_i\} \text{ are disjoint subsets of } \mathbb{N}$$

[1] In the ς-calculus every term is an object.

CALL

$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i(t)\right] \rightarrow_\varsigma E\left[s_i\{x_i \leftarrow [l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}, y_i \leftarrow t\}\right]}$$

UPDATE

$$\frac{l_i \in \{l_j\}^{j\in 1..n}}{E\left[[l_j = \varsigma(x_j, y_j)s_j]^{j\in 1..n}.l_i := \varsigma(x, y)t\right] \rightarrow_\varsigma E\left[[l_i = \varsigma(x, y)t, l_j = \varsigma(x_j, y_j)s_j^{j\in(1..n)-\{i\}}]\right]}$$

LOCAL

$$\frac{s \rightarrow_\varsigma s'}{\alpha[f_i \mapsto s :: Q, t] :: C \rightarrow_\| \alpha[f_i \mapsto s' :: Q, t] :: C}$$

ACTIVE

$$\frac{\gamma \notin (\text{dom } (C) \cup \{\alpha\}) \qquad noFV(s)}{\alpha[f_i \mapsto E[Active(s)] :: Q, t] :: C \rightarrow_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, s] :: C}$$

REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s) \qquad \alpha \neq \beta}{\alpha\,[f_i \mapsto E[\beta.l(s)] :: Q, t] :: \beta[R, t'] :: C \rightarrow_\| \alpha\,[f_i \mapsto E[f_k] :: Q, t] :: \beta\,[f_k \mapsto t'.l(s) :: R, t'] :: C}$$

SELF-REQUEST

$$\frac{f_k \text{ fresh} \qquad noFV(s)}{\alpha\,[f_i \mapsto E[\alpha.l(s)] :: Q, t] :: C \rightarrow_\| \alpha\,[f_k \mapsto t.l(s) :: f_i \mapsto E[f_k] :: Q, t] :: C}$$

REPLY

$$\frac{\beta[f_k \mapsto s :: R, t'] \in \alpha[f_i \mapsto E[f_k] :: Q, t] :: C}{\alpha[f_i \mapsto E[f_k] :: Q, t] :: C \rightarrow_\| \alpha[f_i \mapsto E[s] :: Q, t] :: C}$$

UPDATE-AO

$$\frac{\begin{array}{c}\gamma \notin \text{dom } (C) \cup \{\alpha\} \\ noFV(\varsigma(x, y)s) \qquad \beta[R, t'] \in \alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C\end{array}}{\alpha[f_i \mapsto E[\beta.l := \varsigma(x, y)s] :: Q, t] :: C \rightarrow_\| \alpha[f_i \mapsto E[\gamma] :: Q, t] :: \gamma[\varnothing, t'.l := \varsigma(x, y)s] :: C}$$

Table 1.1: ASP_{fun} semantics

Classically, we define contexts as expressions with a single hole ($\bullet$). $E[s]$ denotes the term obtained by replacing the single hole by s.

$$\begin{array}{rl} E ::= & \bullet \mid [l_i = \varsigma(x, y)E, l_j = \varsigma(x_j, y_j)t_j^{j\in(1..n)-\{i\}}] \mid E.l_i(t) \mid \\ & s.l_i(E) \mid E.l_i := \varsigma(x, y)s \mid s.l_i := \varsigma(x, y)E \mid Active(E) \end{array}$$

The initial configuration contains a single request, the user program:

$$initConf(s_0) = \alpha[f_0 \mapsto s_0, []]$$

where s_0 is a static term without free variable.

Chapter 2

Noninterference

2.1 Introduction

Information flow security [4] and its analysis [5] is the attempt to detect the ways that information flows through the state of a program. Noninterference [6] is the formal notion of confidentiality: the state of a program being compartmented and assigned to security classes must not allow such interference between compartments that would represent a read-up or a write-down. To give a simpler intuition of noninterference: presuming only two classes H (high for confidential) and L (for low or public), an attacker must not learn anything about the information contained in the H variables if he can observe all L variables. The attacker can see all L inputs and outputs of the program on as many runs as he likes, without being able to detect any difference on the L values if the H values change. That is, the L variables are completely independent of the H ones.

The idea to use type systems for a static analysis has first been elaborated by Volpano, Smith and Irvine [19, 18]. For a concise overview of noninterference type systems, the survey by Sabelfeld and Myers is recommended [15]. The work by Banerjee and Naumann introduced the techniques for handling noninterference proofs in dynamic structures that we adapt and clarify for our noninterference proof. Proofs of language semantics and type safety are already complex and very technical; noninterference proofs do not stand back. Thus, following a similar trend for semantics and types, more recently, the analysis of noninterference has triggered work on mechanizations of such security proofs in interactive proof assistants, e.g. [17, 9]. As static analysis of noninterference is incarnated in type systems, the same advantages that have motivated classical type soundness mechanizations remain valid: from a type system mechanized in Isabelle or Coq, executable code can be generated that represents a security analyser (that is moreover certified if corresponding soundness proofs have been conducted in the proof assistant).

The security type system is an extension of the existing type system of

$\mathrm{ASP}_{\mathrm{fun}}$ [8] by security types from the lattice $\langle\{H, L\}, L \leq H\rangle$. The extension is by building pairs of "old" types and security types, L or H. This new type system contains the classical type system thus also guaranteeing deadlock-freedom [8]. We will show subsequently that this slight extension of the original type system additionally proves information flow security. Let us, however first formally introduce the security type system. In Table 2.1, we find the local type system defining inductively the types of local terms. We use the type variable $\delta \in \{L, H\}$ for simple security types and A to denote classical types or structured types, i.e. pairs of classical types and L or H.

$$\textsc{Base}\quad L \leq H \qquad \textsc{Subsumption}\quad \frac{\vdash x : (A, \delta) \qquad \delta \leq \delta'}{\vdash x : (A, \delta')} \qquad \textsc{Val } x\quad x : A :: T \vdash x : A$$

$$\textsc{Type Object}\quad \frac{A = ([l_i : B_i \to D_i]^{i \in 1..n}, \delta) \qquad \forall i \in 1..n,\ x_i : A :: y_i : (B_i, \delta) :: T \vdash t_i : (D_i, \delta)}{T \vdash [l_i = \varsigma(x_i : A, y_i : B_i)t_i]^{i \in 1..n} : A}$$

$$\textsc{Type Call}\quad \frac{T \vdash s : ([l_i : B_i \to D_i]^{\ i \in 1..n}, \delta) \qquad j \in 1..n \qquad T \vdash t : (B_j, \delta)}{T \vdash s.l_j(t) : (D_j, \delta)}$$

$$\textsc{Type Update}\quad \frac{A = ([l_i : B_i \to D_i]^{i \in 1..n}, \delta) \qquad T \vdash s : A \qquad j \in 1..n \qquad x : A :: y : (B, \delta) :: T \vdash t : (D_j, \delta)}{T \vdash s.l_j := \varsigma(x : A, y : B)t : A}$$

Table 2.1: Typing the local calculus

The following preservation theorem can be derived by referring directly to the corresponding theorem for the classical type system [8].

Theorem 1 (Subject Reduction)**.**

$$\vdash C : \langle \Gamma_{act}, \Gamma_{fut} \rangle \ \wedge\ C \to_{\|} C' \Rightarrow \exists\ \Gamma'_{act}, \Gamma'_{fut}.\ \vdash C' : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$$

where $\Gamma_{act} \subseteq \Gamma'_{act}$*, and* $\Gamma_{fut} \subseteq \Gamma'_{fut}$*.*

Proof. The proof of subject reduction for the classical type system stays valid because the noninterference type system just extends the classical type system by new type components L and H. Therefore, the new typing relation can only be more restrictive. If $\vdash_{\text{class}} C : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$ in the classical type system, then $\vdash C : \langle \Gamma^{\star}_{act}, \Gamma^{\star}_{fut} \rangle$ in the new noninterference type system for some extension by L and H. Let E_{act} and E_{fut} such that $E_{act}(\Gamma_{act}) = \Gamma^{\star}_{act}$ and $E_{fut}(\Gamma_{fut}) = \Gamma^{\star}_{fut}$. If now $C \to_{\|} C'$ – due to the classical subject reduction theorem [8] – we have that $\exists\ \Gamma'_{act}, \Gamma'_{fut}.\ \vdash_{\text{class}} C' : \langle \Gamma'_{act}, \Gamma'_{fut} \rangle$. Then we generally have that $\vdash C' : \langle E'_{act}(\Gamma'_{act}), E'_{fut}(\Gamma'_{fut}) \rangle$ where the dashes on the extensions symbolize the continuation by L, H for new elements created by future or activity creation. $\square$

$$\textsc{Type Active}\quad \frac{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash a : A}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \mathit{Active}(a) : A} \qquad \textsc{Type Activity Reference}\quad \frac{\beta \in \mathrm{dom}\ (\Gamma_{act})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash \beta : \Gamma_{act}(\beta)}$$

$$\textsc{Type Future Reference}\quad \frac{f_k \in \mathrm{dom}\ (\Gamma_{fut})}{\langle \Gamma_{act}, \Gamma_{fut}\rangle, T \vdash f_k : \Gamma_{fut}(f_k)}$$

$$\textsc{Type Configuration}\quad \frac{\mathrm{dom}\ (\Gamma_{act}) = \mathrm{dom}\ (C) \qquad \mathrm{dom}\ (\Gamma_{fut}) = \bigcup\{\mathrm{dom}\ (Q) \mid \exists\ \alpha, a.\ \alpha[Q,a] \in C\} \qquad \forall\, \alpha[Q,a] \in C.\ \begin{cases}\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash a : \Gamma_{act}(\alpha) & \wedge \\ \forall\, f_i \in \mathrm{dom}\ (Q).\ \langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash Q(f_i) : \Gamma_{fut}(f_i)\end{cases}}{\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle}$$

Table 2.2: Typing configurations

2.2 Indistinguishability and Noninterference

In this section, we show now that any well-typed program is secure. To this end, we first define a relation of indistinguishability, often also called L-equivalence because in this relation L-terms have to be equal.

To improve the exposition, we refer in the following only to the L/H components of the types in all typing statements – without explicitly projecting out the classical type constituents.

Lemma 2.2.1 (Confinement)**.** *Let* $\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle$ *and* $\alpha[Q,t] \in C$ *with* $\Gamma_{act}(\alpha) = \delta$ *for* $\delta \in D$*. Then, we have that* $\forall\, f_k \in \mathrm{dom}\ (Q).\ \Gamma_{fut}(f_k) = \delta$*.*

Proof. Since $\vdash C : \langle \Gamma_{act}, \Gamma_{fut}\rangle$ it follows from Type Configuration for $(f_k, s) \in Q$, that $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash s : \Gamma_{fut}(f_k)$. By Type Future Reference we also have that $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash f_k : \Gamma_{fut}(f_k)$. That is, s and f_k have the same type. As a consequence of the semantics for s, there exists s_0 with $s_0 \rightarrow_{\|}^{*} s$ and $s_0 = t.l(p)$ for some former call to α that resulted by rule request to this $t.l(p)$. By rule Type Call and Type Object, we know that $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash t.l(p) : \delta$ if $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash t : \delta$ which in turn follows by Type Configuration from $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash \alpha : \delta$ which we know by assumption. By preservation $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash s : \delta$ because $\langle \Gamma_{act}, \Gamma_{fut}\rangle, \varnothing \vdash s_o : \delta$ and $s_0 \rightarrow_{\|}^{*} s$. As we have seen earlier, the type of s is equal to the type of f_k and thus to the type of α. Since $f_k \in \mathrm{dom}\ (Q)$ was arbitrary, we are finished. □

Definition 2.2.2 (Typed Bijection)**.** *A typed bijection is a finite partial function*

σ on activities α (or futures f_k respectively) such that

$$\forall\, a : \text{dom}\ (\sigma).\ \vdash a : T\ \Rightarrow \vdash \sigma(a) : T$$

(where T is given by $\Gamma_{act}(a)$ or $\Gamma_{fut}(a)$ respectively).

The intuition behind typed bijections is that dom (σ) designates all those futures or activity references that are or have been visible to the attacker. We cannot assume the names in different runs of programs, even for low elements, to be the same. Hence, we relate those names via a pair of bijections. These bijection are typed because they relate activities and futures that are all of type L. The following definition of indistinguishability uses the typed bijection in this sense. The intuitive relationship between type L and membership in dom (σ) is only later made formal by an invariant. We believe that this invariant decisively ameliorates the exposition of the proofs and the understanding of the model (compare with the proofs in Banerjee and Naumann's paper [1]).

We define (low)-indistinguishability as a relation $\sim_{\sigma,\tau}$ parameterized by two typed bijections one over activity names and one over futures. It is a heterogeneous relation as it ranges over elements of different types, for example activities and request queues. We leave out the types as they are indicated by our notational convention. By $T_{\sigma,\tau}$ we denote the term (or type) T where all occurrences of activity names a or futures f are replaced by their counterparts $\sigma(a)$ or $\tau(f)$, respectively, given they are in the domain, otherwise unchanged.[1]

Definition 2.2.3 (Indistinguishability)**.** *An indistinguishability relation is a heterogeneous relation $\sim_{\sigma,\tau}$, parameterized by two isomorphisms σ and τ whose differently typed subrelations are as follows.*

$$\begin{aligned}
t \sim_{\sigma,\tau} t' &\equiv t_{\sigma,\tau} = t' \\
\alpha_0 \sim_{\sigma,\tau} \alpha_1 &\equiv \tau(\alpha_0) = \alpha_1 \\
f_k \sim_{\sigma,\tau} f_j &\equiv \sigma(f_k) = f_j \\
[R_{\alpha_0}, t_{\alpha_0}] \sim_{\sigma,\tau} [R_{\alpha_1}, t_{\alpha_1}] &\equiv R_{\alpha_0} \sim_{\sigma,\tau} R_{\alpha_1} \wedge t_{\alpha_0} \sim_{\sigma,\tau} t_{\alpha_1} \\
R_{\alpha_0} \sim_{\sigma,\tau} R_{\alpha_1} &\equiv \begin{array}{l} \text{dom}\ (\sigma) \subseteq \text{dom}\ (R_{\alpha_0}) \wedge \text{ran}\ (\sigma) \subseteq \text{dom}\ (R_{\alpha_1}) \wedge \\ \quad \forall\, f_k, f_j\ .f_k \sim_{\sigma,\tau} f_j \Rightarrow R_{\alpha_0}(f_k) \sim_{\sigma,\tau} R_{\alpha_1}(f_j) \end{array} \\
C_0 \sim_{\sigma,\tau} C_1 &\equiv \begin{array}{l} \text{dom}\ (\tau) \subseteq \text{dom}\ (C_0) \wedge \text{ran}\ (\tau) \subseteq \text{dom}\ (C_1) \wedge \\ \forall\, \alpha_0, \alpha_1.\ \alpha_0 \sim_{\sigma,\tau} \alpha_1 \Rightarrow C_0(\alpha_0) \sim_{\sigma,\tau} C_1(\alpha_1) \end{array}
\end{aligned}$$

We repeat here the remark made by the designers of this kind of indistinguishability definition [1]: "The above exploits the convention that equations involving partial functions are interpreted as false when the function is undefined." Thus, α_0, α_1 (or f_k, f_j, respectively) are in the relation $\sim_{\sigma,\tau}$ if

[1] For types Γ_{fut} and Γ_{act}, the names are replaced in the domain which produces a somewhat "contravariant" effect visible in the direction of the replacement.

and only if (α_0, α_1) is in τ (or $(f_k, f_j) \in \sigma$, respectively) because otherwise $(\tau(\alpha_0) = \alpha_1) = \mathit{false}$ (or $(\sigma(f_k) = f_j) = \mathit{false}$). In case of $\alpha_0 \notin \text{dom } (\tau)$, for example, $C(\alpha_0) \sim_{\sigma,\tau} C(\alpha_1)$ could be true or false illustrating the partiality of the definition of indistinguishability. The entire high part of the program is not relevant for L-indistinguishability and thus not recorded at all in the corresponding typed bijections. That is, "H-indistinguishability" really corresponds to "indistinguishability not defined".

The partial bijection approach is an elegant concept for specification but technically proofs become hard to follows. Therefore, we explicitly mark the correspondence between type L and domains of isomorphisms. The following invariant specifies this correspondence.

Definition 2.2.4 (Invariant)**.**

$$\begin{aligned} \alpha_0 \in \text{dom } (\tau) &\equiv \Gamma_{act}(\alpha_0) = L \\ f_k \in \text{dom } (\sigma) &\equiv \Gamma_{fut}(f_k) = L \end{aligned}$$

We write invariant(σ, τ) if the configurations are clear from context.

The invariant immediately transfers to the ranges of σ and τ because they are typed bijections.

Corollary 2.2.5. *If the invariant holds we also have the following equivalences.*

$$\begin{aligned} \alpha_1 \in \text{ran } (\tau) &\equiv \Gamma_{act}(\alpha_1) = L \\ f_j \in \text{ran } (\sigma) &\equiv \Gamma_{fut}(f_j) = L \end{aligned}$$

Note, that the invariant only specifies this correspondence. The invariant is a tool to clarify the proof of noninterference. Its validity for given typings and pairs of configurations has to be established.

To ground the indistinguishability we define when two initial configurations are indistinguishable.

Definition 2.2.6 (Initial Indistinguishability)**.** *Two well-typed initial configurations initConf(s_0) and initConf(s_1) are indistinguishable ($\sim$) if either $s_0 = s_1$ or $\Gamma_{act}(\alpha_0) = \Gamma_{act}(\alpha_1) = H$.*

Using this idea of initial indistinguishability, we can establish that the invariant holds for initial configurations if they are indistinguishable.

Lemma 2.2.7 (Initial Invariant)**.** *Given two indistinguishable initial configurations that are well-typed, the isomorphisms σ and τ can be constructed such that the invariant holds.*

$$\forall\, s_0, s_1. \exists\, \sigma, \tau. \begin{cases} \langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash \mathit{initConf}(s_0) : \Gamma_{act}(\alpha_0) \\ \langle \Gamma_{act}, \Gamma_{fut} \rangle_{\sigma,\tau}, \varnothing \vdash \mathit{initConf}(s_1) : \Gamma_{act_{\sigma,\tau}}(\alpha_1) \Rightarrow \mathit{invariant}(\sigma, \tau) \\ \mathit{initConf}(s_0) \sim \mathit{initConf}(s_1) \end{cases}$$

Proof. Let $initConf(s_0)$ and $initConf(s_1)$ be two well-typed indistinguishable initial configurations. If $s_0 = s_1$ and $\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash s_0 : \Gamma_{fut}(f_0)$ then $\langle \Gamma_{act}, \Gamma_{fut} \rangle, \varnothing \vdash s_1 : \Gamma_{fut}(f_0)$ but also $\langle \Gamma_{act}, \Gamma_{fut} \rangle_{\sigma,\tau}, \varnothing \vdash s_1 : \Gamma_{fut_{\sigma,\tau}}(f_1)$. Hence, by type uniqueness $\Gamma_{fut}(f_0) = \Gamma_{fut_{\sigma,\tau}}(f_1)$ and by the Confinement lemma $\Gamma_{act}(\alpha_0) = \Gamma_{act_{\sigma,\tau}}(\alpha_1)$. If both sides are H, we are in the second case (see below). If they are L, set $\tau(\alpha_0) = \alpha_1$ and $\sigma(f_0) = \sigma(f_1)$. Then, τ and σ are well-defined typed bijections and the Invariant holds. If now, for the second case, $\Gamma_{act}(\alpha_0) = \Gamma_{act}(\alpha_1) = H$ then set $\tau = \sigma = \varnothing$. Then, again, the two maps are well-defined and the Invariant holds. □

Corollary 2.2.8. *The isomorphisms σ, τ constructed in the proof of the previous lemma additionally turn the initial configuration indistinguishable in the sense of Definition 2.2.3.*

$$initConf(s_0) \sim initConf(s_1) \Rightarrow initConf(s_0) \sim_{\sigma,\tau} initConf(s_1)$$

Note, that if the initial configurations were not indistinguishable, their types could be different in which case the existence of a pair of isomorphisms fulfilling the invariant would be impossible. This lemma will be used to enable – in the final Corollary 2.2.9 – the following main theorem proving the noninterference of well-typed configurations. The main theorem assumes the invariant as a hypothesis and shows that it is preserved.

Note, furthermore, that we prove a strong version of bisimulation in which the second transition is $\rightarrow_{\parallel}^{01} = id \cup \rightarrow_{\parallel}$ and not just $\rightarrow_{\parallel}^{*}$ (as, for example, in [16]).

Theorem 2 (Noninterference)**.** *Let C_0 and C_1 be configurations such that $C_0 \sim_{\sigma,\tau} C_1$, $\vdash C_0 : \langle \Gamma_{act}, \Gamma_{fut} \rangle$ and $\vdash C_1 : \langle \Gamma_{act}, \Gamma_{fut} \rangle_{\sigma,\delta}$, and the Invariant holds. If $C_0 \rightarrow_{\parallel} C_0'$ then there exists C_1' such that $C_1 \rightarrow_{\parallel}^{01} C_1'$ and $C_0' \sim_{\sigma',\tau'} C_1'$ such that $\sigma \subseteq \sigma'$, $\tau \subseteq \tau'$, and the invariant remains valid for σ' and τ'.*

Proof. We proceed by case analysis and induction over the semantics $\rightarrow_{\parallel}$. In each case, we define new σ' and τ' based on the existing σ and τ for which the invariant holds by assumption. The case analysis hinges on $\alpha \in \text{dom}\ (\tau)$ and $f \in \text{dom}\ (\sigma)$ rather than L and H as in classical proofs, e.g. [16] (however, it is important to keep in mind that this predicate corresponds to H/L-typing in form of the proof invariant).

The **high** case is proved once for all cases of the semantic reduction. Let $\alpha_0 \in \text{dom}\ (C_0)$ and $\alpha_0 \notin \text{dom}\ (\tau)$ with $C_0(\alpha_0) \neq C_0'(\alpha_0)$, i.e. this activity has been reduced. Let $\sigma' = \sigma$, $\tau' = \tau$, and $C_1' = C_1$. Then, $C_0' \sim_{\sigma,\tau} C_1'$ because $\text{dom}\ (\tau' = \tau) \subseteq \text{dom}\ (C_0) \subseteq \text{dom}\ (C_0')$ and similarly for σ and C_1. The new activities that may have been introduced in case the reduction was with rules ACTIVE or UPDATE-AO are H since by the invariant from $\alpha_0 \notin \text{dom}\ (\tau)$ follows that $\Gamma_{act}(\alpha_0) = H$. In turn, by preservation and confinement, the new activities have type H whereby the invariant remains valid and indistinguishability as well.

The other case $\alpha_0 \in \text{dom}\,(C_0)$ such that $\alpha_0 \in \text{dom}\,(\tau)$ and $C_0(\alpha_0) \neq C_0'(\alpha_0)$ entails the **low** cases which are proved case by case following the semantics. Generally, we know as $\alpha_0 \in \text{dom}\,(\tau)$ that $\Gamma_{act}(\alpha_0) = L$ and that $\tau(\alpha_0) = \alpha_1$ for some $\alpha_1 \in \text{dom}\,(C_1)$. Furthermore, $\Gamma_{act_{\sigma,\tau}}(\alpha_1) = L$ because τ preserves types, and all contained futures are L by confinement. Let for all cases $C_0(\alpha_0) = \alpha_0\,[R_0, t_0]$ and $C_1(\alpha_1) = [R_1, t_1]$.

Case 1 (LOCAL). Since $C_0 \sim_{\sigma,\tau} C_1$, we have $R_0 \sim_{\sigma,\tau} R_1$ which means that $\forall\, f_k, f_j.\ f_k \sim_{\sigma,\tau} f_j \Rightarrow R_0(f_k) \sim_{\sigma,\tau} R_1(f_j)$. As the reduction was with LOCAL, we have an f_k with $R_0(f_k) = E[s]$ and $s \rightarrow_\varsigma s'$. Because of the Invariant, we have $\Gamma_{act}(\alpha_0) = L$ and because of confinement $\Gamma_{fut}(f_k) = L$. Hence, using the Invariant in the opposite sense, we have $f_k \in \text{dom}\,(\sigma)$, i.e. $\sigma(f_k) = f_j$ with $R_0(f_k) \sim_{\sigma,\tau} R_1(f_j)$ which means (according to indistinguishability for terms) that $R_0(f_k)_{\sigma,\tau} = R_1(f_j)$. That is $R_0(f_k) = E(s)$ and hence $R_1(f_j) = E_{\sigma,\tau}[s_{\sigma,\tau}]$. Since $s \rightarrow_\varsigma s'$, also $s_{\sigma,\tau} \rightarrow_\varsigma s'_{\sigma,\tau}$ and since, clearly, $s'_{\sigma,\tau} = s'_{\sigma,\tau}$, we have $E[s] \sim_{\sigma,\tau} E_{\sigma,\tau}[s_{\sigma,\tau}]$. We can thus set C_1' with R_1' updated to $R_1(f_j \mapsto E_{\sigma,\tau}[s'_{\sigma,\tau}])$. Then, $C_1 \rightarrow_\parallel C_1'$, and $C_0' \sim_{\sigma,\tau} C_1'$. The typed bijections remain the same and – as there are no new L elements introduced – consequently the Invariant remains valid.

Case 2 (ACTIVE). Let $C_0(\alpha_0) = \alpha_0[R_0, t_0]$ with $R_0(f_k) = E[\mathit{Active}(s)]$, $C_0'(\beta_0) = \beta_0[\varnothing, s]$, and $C_0'(\alpha_0) = \alpha_0[R_0', t_0]$ such that $R_0'(f_k) = E[\beta_0]$ according to semantic rule ACTIVE. Similar to the previous case, since we consider $\alpha_0 \in \text{dom}\,(\tau)$, we have, due to the Invariant, that $\Gamma_{act}(\alpha_0) = L$ and consequently (confinement) $\Gamma_{fut}(f_k) = L$ which again gives, by invariant(σ, τ) that $f_k \in \text{dom}\,(\sigma)$. In turn, we know that there are α_1 and f_j with $(\alpha_0, \alpha_1) \in \tau$ and $(f_k, f_j) \in \sigma$ such that – due to indistinguishability – $C_1(\alpha_1) = \alpha_1[R_1, t_1]$ with $R_1(f_j) = E_{\sigma,\tau}[\mathit{Active}(s_{\sigma,\tau})]$. We can select $C_1' = \alpha_1[R_1(f_j \mapsto E_{\sigma,\tau}[\beta_1]), t_1] \cup \beta_1[\varnothing, s_{\sigma,\tau}]$. Then, $C_1 \rightarrow_\parallel C_1'$ by rule ACTIVE as well. According to preservation, the successor configurations are well-typed with types $\langle \Gamma_{act}{}', \Gamma_{fut}{}' \rangle = \langle \Gamma_{act}(\alpha_1 \mapsto \mathit{type}(s)), \Gamma_{fut} \rangle$ and with subscript σ', τ' for C_1 by typing rule TYPE ACTIVE. The new activities β_0 and β_1 have type L by confinement and TYPE ACTIVE; the new isomorphism is $\tau' = \tau \cup (\beta_0, \beta_1)$ for activities; the future isomorphism stays the same, $\sigma' = \sigma$. Thereby, the Invariant remains true for σ' and τ'. Finally, we see that $C_0' \sim_{\sigma,\tau} C_1'$.

Case 3 (REQUEST). Let for $\alpha_0[R_0, t_0]$, according to the proviso of REQUEST, $R_0(f_k) = E[\beta_0.l(s)]$ and $C_0(\beta_0) = \beta_0[R_{\beta_0}, t_{\beta_0}]$. From confinement and local typing rules, we know that types of α_0 and β_0 are equal. Since by invariant, from $\alpha_0 \in \text{dom}\,(\tau)$, follows that $\Gamma_{act}(\alpha_0) = L$ then so is $\Gamma_{act}(\beta_0)$. In turn, $\beta_0 \in \mathit{dom}(\tau)$ by $\mathit{invariant}(\sigma, \tau)$. In C_0' we have $R_0'(f_k) = E[f_m]$, for some fresh f_m and $R'_{\beta_0} = R_{\beta_0}(f_m \mapsto t_{\beta_0}.l(s))$. By confinement and preservation, we know that $\Gamma_{fut}{}'(f_m)$ must be L. Thus, there exist α_1, β_1 with $\{(\alpha_0, \alpha_1), (\beta_0, \beta_1)\} \subseteq \tau$, and f_j, f_h with $\{(f_k, f_j), (f_m, f_h)\} \subseteq \sigma$ such that $C_1(\alpha_1) = \alpha_0[R_1, t_1]$, $C_1(\beta_1) = \beta_1[R_{\beta_1}, t_{\beta_1}]$. From $R_0(f_k) \sim_{\sigma,\tau} R_1(f_j)$ follows that $R_1(f_j) = E_{\sigma,\tau}[\beta_1.l(s_{\sigma,\tau})]$. Now, set C_1' such that $R_1' = R_1(f_j \mapsto E_{\sigma,\tau}[f_h])$

and $R_{\beta_1} = R_{\beta_1}(f_h \mapsto t_{\beta_1}.l(s_{\sigma,\tau}))$. Then, $C_1 \rightarrow_{\|} C_1'$ by rule REQUEST. Set $\sigma' = \sigma \cup \{(f_m, f_h)\}$ and $\tau' = \tau$. Now, $C_0' \sim_{\sigma,\tau} C_1'$ and *invariant*(σ', τ') is valid.

Case 4 (SELF-REQUEST). This case is very similar to the previous case REQUEST by simply using $\alpha_0 = \beta_0$, and $\alpha_1 = \beta_1$.

Case 5 (REPLY). In this case, we have $C_0(\alpha_0) = \alpha_0[R_0, t_0]$ with $R_0(f_k) = E[f_m]$ and $C_0(\beta_0) = \beta_0[R_{\beta_0}, t_{\beta_0}]$ with $R_{\beta_0}(f_m) = s$. According to $\alpha_0 \in \text{dom}\ (\tau)$, confinement, and typing rules, we have that $\Gamma_{act}(\alpha_0) = \Gamma_{act}(\beta_0) = L$ and $\beta_0 \in \text{dom}\ (\tau)$, again, by Invariant. In C_0', we have – according to semantic rule REPLY – $R_0'(f_k) = E[s]$, $R'_{\beta_0} = R_{\beta_0}$ unchanged, and the type $\Gamma_{fut}(f_k)$ is the type of s and equally for f_m. As before, $\alpha_0 \in \text{dom}\ (\tau)$ gives that $\Gamma_{act}(\alpha_0) = L$ by *invariant*(σ, τ). Hence, there exist α_1, β_1 with $\{(\alpha_0, \alpha_1), (\beta_0, \beta_1)\} \subseteq \tau$, and f_j, f_h with $\{(f_k, f_j), (f_m, f_h)\} \subseteq \sigma$ giving us a similar situation in configuration C_1 as $C_1(\alpha_1) = \alpha_1[R_1, t_1]$ with $R_1(f_j) = E_{\sigma,\tau}[f_h]$ and $C_1(\beta_1) = \beta_1[R_{\beta_1}, t_{\beta_1}]$ with $R_{\beta_1}(f_h) = s_{\sigma,\tau}$. Now, we can select C_1' according to semantic rule (REPLY) such that $R_1'(f_j) = E[s_{\sigma,\tau}]$, $R'_{\beta_1} = R_{\beta_1}$; hence $C_1 \rightarrow_{\|} C_1'$. Set $\tau' = \tau$ and $\sigma' = \sigma$, then $C_0' \sim_{\sigma,\tau} C_1'$ because $R_0'(f_k) \sim_{\sigma,\tau} R_1'(f_j)$ and $R'_{\alpha_0}(f_m) \sim_{\sigma,\tau} R'_{\alpha_0}(f_h)$.

Case 6 (UPDATE-AO). For $C_0 \rightarrow_{\|} C_0'$ with UPDATE-AO, let $C_0(\alpha_0) = \alpha_0[R_0, t_0]$, $C_0(\beta_0) = \beta_0[R_{\beta_0}, t_{\beta_0}]$ and $R_0(f_k) = E[\beta_0.l := \varsigma(x, y)s]$. Then, following the semantics, the post-state is $R_0'(f_k) = E[\gamma_0]$, $C_0'(\gamma_0) = \gamma_0[\varnothing, t_{\beta_0}.l := \varsigma(x, y)s]$ for some fresh γ_0. Again, we have since α_0 and, consequently, $\beta_0 \in \text{dom}\ (\tau)$ that they are L by Invariant. Thus there are, in C_1, corresponding names α_1, β_1, f_j with $\{(\alpha_0, \alpha_1), (\beta_0, \beta_1)\} \subseteq \tau$ and $(f_k, f_j) \in \sigma$ and $C_1(\alpha_1) = \alpha_1[R_1, t_1]$, $C_1(\beta_1) = \beta_1[R_{\beta_1}, t_{\beta_1}]$. Since $R_0 \sim_{\sigma,\tau} R_1$ and $t_{\alpha_1} =_{\sigma,\tau} t_{\beta_1}$, we have $R_1(f_j) = E_{\sigma,\tau}[\beta_1.l := \varsigma(x, y)s_{\sigma,\tau}]$, we can set C_1' – consistent with the semantics – such that $R_1'(f_j) = E_{\sigma,\tau}[\gamma_1]$, $C_1'(\gamma_1) = \gamma_1[\varnothing, t_{\beta_1}.l := \varsigma(x, y)s_{\sigma,\tau}] = \gamma_1[\varnothing, t_{\alpha_1\sigma,\tau}.l := \varsigma(x, y)s_{\sigma,\tau}]$, hence $C_1 \rightarrow_{\|} C_1'$. Set $\tau' = \tau \cup \{(\gamma_0, \gamma_1)\}$, $\sigma' = \sigma$. The type $\Gamma_{act}(\gamma_0) = \Gamma_{act}(\gamma_1) = L$ since $t_{\alpha_1}, t_{\beta_1}$ are L and thus the updates as well. Thus, with the extension, the Invariant stays valid, and we can conclude, again, $C_0' \sim_{\sigma,\tau} C_1'$. □

Corollary 2.2.9 (Noninterference of Reachable Configurations)**.** *Let C_0 and C_1 be configurations reachable from some initial indistinguishable configurations then there exist σ and τ such that $C_0 \sim_{\sigma,\tau} C_1$.*

The corollary follows by induction over $\rightarrow_{\|}$ from Lemma 2.2.7, Corollary 2.2.8, and the Noninterference Theorem.

2.3 Conclusions

We now endeavour to interpret the formal presentation given in the paper. To this end, we summarize some lessons learnt, discuss some related work to finally conclude with possible alternatives that might improve the practical relevance of the presented result and thus the applicability of ASP_{fun} for security-critical systems.

2.3.1 Lessons Learnt

The noninterference type system may seem overly restrictive: activities can only communicate with other activities if they have exactly the same security class. One might think that the typing rules can be defined in a more liberal way: an activity should be allowed to receive replies from activities at a lower level and activities should be able to utter requests to a higher level. This would represent a *write up* and a *read down*, respectively, which are consistent with the *no read up* and *no write down* idea. However, we omitted this completely from the exposition – also to facilitate the reasoning – but as we believe rightly for the following reason. A request is only uttered to receive a corresponding reply. That is, method call in ASP_{fun} is only complete in the combination request plus reply. But then, clearly, the two restrictions pointed out before enforce the requesting and the receiving activity to be on the same level. This is a well-known problem from security in distributed systems.

Our approach follows a method for defining and proving noninterference that stems from the application to imperative programs [16, 1]. Since ASP_{fun} is a functional language, we simply interpret data values as the terms contained in the request list of an activity. This reflects that – although functional – ASP_{fun} activities contain the current state of evaluation. Consequently, the initial configuration represents the input to an ASP_{fun} program. The output are the terms that remain in the future lists of the activities when the activities are fully evaluated. Therefore, a security policy for ASP_{fun} is given by the initial setting of the security level for the term in an initial configuration. (The fact that we define indistinguishability for the initial configuration represents the gist of the security policy: a legal security policy assigns the security values such that for L-indistinguishable initial values no information is leaked).

The fact that activities may only communicate with others that are exactly on their level means, practically, that a remote activity can only interact with foreign activities in its remote domain if they are of the same class.

In some sense this makes the L class equivalent to the H class and fails to correspond to the Multi-Level-Security model (MLS) [4] where information may flow up according to some hierarchical security lattice. In another sense, the strictness corresponds to the needs of a distributed object community. When considering privacy, we enforce thereby that activities are only accessible to other activities of the same private class. But then we need to have a separate class for each user. The typing and the related noninterference result remain valid. Technically, the argument has to be interpreted as H is the class of this user, and the L-class represents the rest of the world. In fact, this is the right assumption for a privacy oriented model: we assume the worst case, that the entire surrounding system is corrupted and exchanges information. Trusted object communities can only be those that are proved to be noninterfering with the rest of the hostile community and this is the same as that the trusted community is also of this users H-class.

2.3.2 Related Work

The approach by Boudol and Castellani is, similar to ours, based on the work by Volpano, Smith, and Irvine [19]. As in the paper [16], Boudol and Castellani address the problem that concurrent while loops can implement implicit flows from the while-control variable x to other observing processes which in case that x is H (and the observer is L) represents an implicit flow. They improve Volpano and Smith's original very restrictive type system that forbids H in while-control by a more fine-grained approach. The main difference of both works to our language is that they are concerned with access to shared data which we completely ignore as we assume separate data spaces for activities. Inside our activities the concurrent threads represented by future values are functional and have no side effects.

Sabelfeld and Mantel define a type-based approach for distributed programs [13]. Their approach differs in that it is entirely concentrated on channels and their security types. There are many similarities, they also consider asynchronous communication, for example, but in our model the message passing abstracts from channels. Even regarding their explicit approach to channels with encryption and devices for detection of timing leaks, we still could not find an answer to the obvious – and well-known, as sketched above – problem that in distributed systems secure communication quickly becomes restricted to one class.

Concerning the ς-calculus we need to mention the early work by Barthe and Serpette [3].

None of the above mentioned works is formalised in an interactive theorem prover. We believe – and might have convinced the reader with the exposition of the noninterference proof – that type systems and safety proofs, in particular noninterference proofs, deserve a rigorous mechanical verification.

Concerning related mechanized noninterference work, there are several interesting works. We would like to mention a couple that are close to our work presented in this paper. An Isabelle/HOL formalisation exists by Naumann of a portion of their imperative language [14]. Concerning parallel languages, the work by Barthe and Prensa-Nieto presents an Isabelle/HOL formalisation of a parallel language [2].

2.3.3 Future Work

We plan to formalise the security model and proofs in Isabelle/HOL based on the complete formalisation of ASP_{fun} in this interactive theorem prover.

One point we would like to investigate is whether the semantics can be slightly adapted to enable a list of initial activities, or, an initial configuration with a list of initial terms. This would enable to experiment with activities of different classes. We see the possibilities worth investigating as follows.

- We could allow several activities in an initial configuration. These would be similar to the single one in the current definition, i.e. contain no references to futures and other activities. Thereby, the semantics would not

change much, previous results remain valid, and the noninterference proof can be also adapted. The definition of initial indistinguishability would need to be scaled up but this represents no problem. Advantage is that the changes to model and proofs are small, but it is not clear whether this change would bring about the desired generality. Although we can have now various activities of type H and L, initially, there are no references in the initial activities, so the question remains, how can activities know of each other so they can communicate – provided they are in the same class. To this end, we would need some kind of global name registry, but then names would become first class citizens of the language. Although names are already part of the language as activity references, the name registry representation implies a richer model where names have some sort of specification attached to it.

- Scaling down the problem of several differently typed elements in the initial configuration, the next possibility is to have security classes only for futures and not for activities and enabling several possibly differently typed futures in the request list of the initial configuration. The problem, here, is again the name spacing; we equally need some register if we want to exclude references in the initial future values. Concerning proofs we loose the confinement property at the activity level, but might regain one at something like the "future level".

- One level zoomed in again, we could envisage that various parts of a ς-term in a future could have different types. The name registry problem remains, and the confinement question becomes even more critical. We might now establish some sort of "thread" confinement.

In all cases a name registry seems to be necessary to realize non-trivial communication amongst different tribes, i.e. families of object with equal security class, in the active object community. The further we zoom into the terms following the previous three approaches, it becomes more likely that this registry might even be a first-class citizen of $\mathrm{ASP}_{\mathrm{fun}}$, i.e. programmed as an activity itself. However, at the same time, the more we zoom in, the more proofs deviate from what we have presented here.

It is also worth investigating whether the view that several security classes exist can be made explicit in our model. Although the informal privacy argument in the previous subsection is clear enough, it is just technically interesting to see how partial bijections can be scaled up to more than just L and H. Note, that the classical technique of "undefined for H" does not simply scale up. Although not relevant for the restrictive security model of $\mathrm{ASP}_{\mathrm{fun}}$ presented in this paper, it might be interesting for more liberal approaches that use a nontrivial security lattice, like the Decentralized Label Model (DLM) by A. C. Myers [12].

Chapter 3

Distributed Consensus

3.1 Paxos

The Paxos algorithm, e.g. [10, 11] is a classical algorithm for the implementation of the distributed consensus problem. That is, it is an algorithm that enables a common vote amongst distributed processes that communicate asynchronously. In this note, we very briefly sum up the algorithm in its simplest form. We then present an implementation of Paxos in ASP_{fun}[8]. This practical implementation serves to illustrate the use of ASP_{fun}, leading into a discussion of the capacities and characteristics of ASP_{fun} as a distributed computation model.

We first introduce the functioning by listing some important properties of the algorithm followed by its protocol.

- Communication is asynchronous, messages may be delayed, lost but not changed.
- Possible rôles are leader (proposer), acceptor, and learner; any agent/process can act in any of those three rôles.
- The Paxos algorithm is a protocol in two phases:
  - A proposer first *proposes* to a group of acceptors a number n representing a possible vote; propositions are numbered in increasing order
  - Acceptors *promise* to stick to that possible vote n unless they have already promised a larger vote $m > n$
  - If the proposer receives a majority (quorum) of positive promises he asks in phase_2 the majority to finally accept vote (n, v) for some v.
- Acceptors accept any vote (unless its number is too low).

For simplicity, we initially assume that proposer and learner are identical and are unique (for an important complication caused by several proposers see Section 3.3).

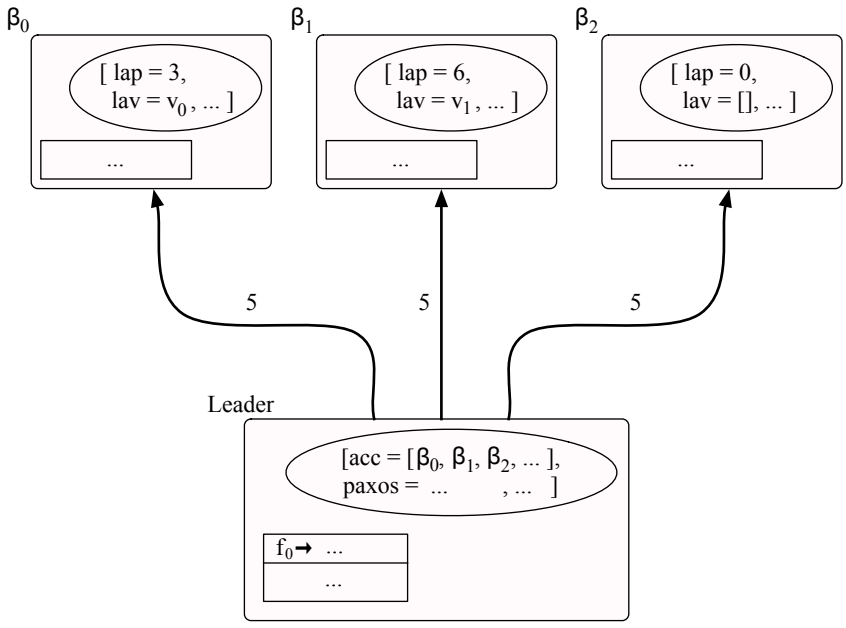

Figure 3.1: Phase 1: leader proposes to three acceptors in different states.

Then, the Paxos algorithm consists of two phases, comprising message exchanges between a leader Λ and a community of acceptors α_i, where $i \in \mathbb{N}$.

$$
\begin{array}{lll}
\text{phase}_1: & \Lambda \to \alpha_i: & \text{proposition n} \\
 & \alpha_i \to \Lambda: & \begin{cases} (n, \langle\rangle_{\text{val}}) & \text{if n is largest proposition and} \\ & & \text{no value accepted so far} \\ (m, v) & \text{if n is largest proposition and} \\ & & v \text{ accepted with } m \leq n \\ \text{Nack} & n < \text{last accepted proposal} \end{cases} \\
\text{phase}_2: & \Lambda \to \alpha_i: & \text{accept!}(n, v) \\
 & \alpha_i \to \Lambda: & \begin{cases} \text{accepted}(n, v) & \text{if last accepted proposal} \quad \leq n \\ \text{Nack} & n < \text{last accepted proposal} \end{cases}
\end{array}
$$

Note, that – in principle – an acceptor has to accept any vote that is proposed to him. An intuition for the use of Paxos is that the proposer acts as a server interface to some external client asking him to coordinate a group of processes, for example, to do a distributed database commit.

The illustrations given in Figures 3.1 to 3.4 illustrate the Paxos algorithm – the variables used in the graphics are introduced in the following section.

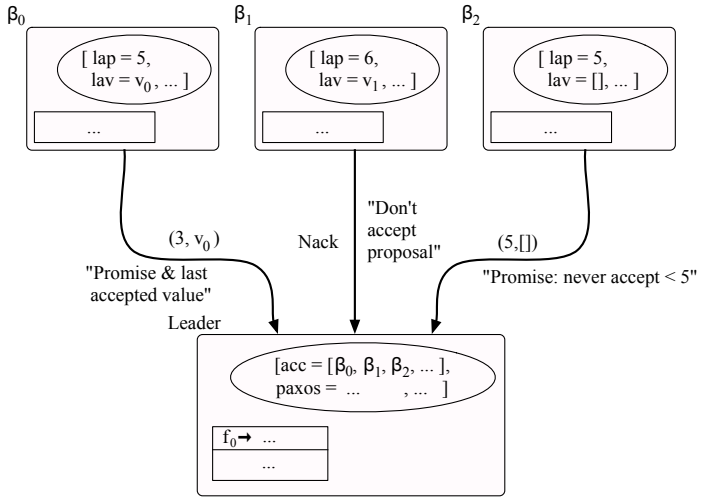

Figure 3.2: Phase 1: one acceptor promises with no prior value, one refuses due to prior acceptance of higher proposition, and one promises by returning last accepted value.

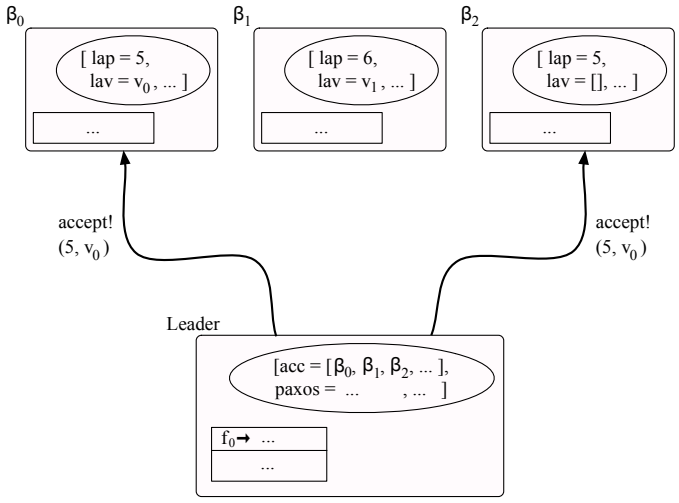

Figure 3.3: Phase 2: leader asks those that have promised to accept highest value.

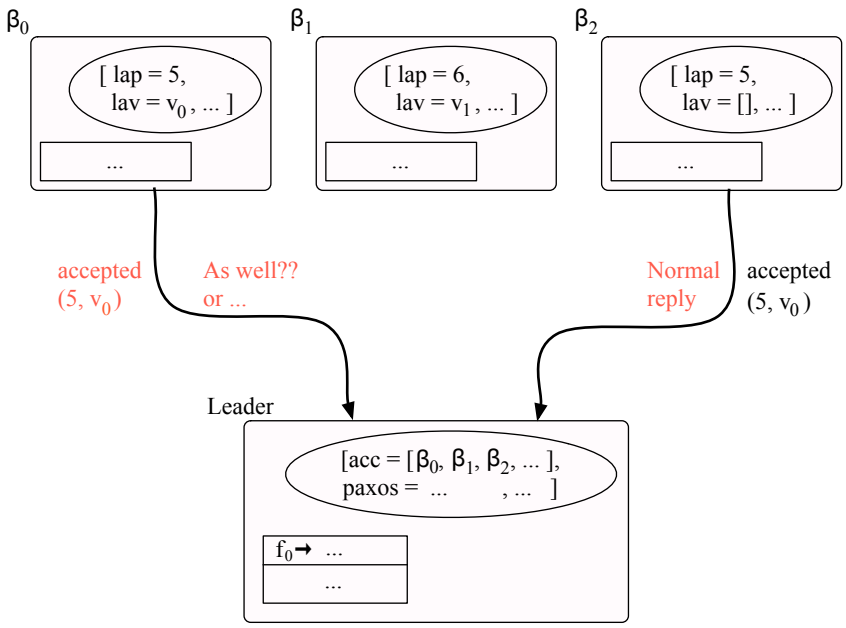

Figure 3.4: Phase 2: the quorum accepts, unless . . . (see Section 3.3.1)

3.2 Paxos in $\mathrm{ASP}_{\mathrm{fun}}$

We present the Paxos algorithm in $\mathrm{ASP}_{\mathrm{fun}}$ by first introducing the agent object, a common type of object that can be used to represent proposers and acceptors alike. That is, the agent contains all necessary functionality for proposing and accepting votes. In particular, an agent object contains a list of activity references called "acceptors" representing the community of agents that can be asked to vote on a value.

$$\text{agent} : [\text{acceptors} = \langle \alpha, \beta, \gamma, \ldots \rangle, \text{paxos}_{\text{fun}} = \ldots, \ldots]$$

We assume as given a few basic data structures. For example lists, are denoted as above, using brackets $\langle$ and $\rangle$, the emtpy list $\langle\rangle_{\text{list}}$, their construction as $x :: l$, and the append operator for lists is @. Furthermore, we use the datastructures of booleans, natural numbers, pairs, a type of values including an empty value denoted as $\langle\rangle_{\text{val}}$. The empty active object is denoted by $\langle\rangle_{\text{obj}}$.

We next assume a set of *agent* objects, an already chosen leader Λ such that the basic configuration for the Paxos algorithm initially is something like the following, where Λ is the leader activity; an initial call to the Paxos algorithm – implemented in each agent as a method – is placed on the request queue of this (unique) leader Λ as future f_0.

$$\begin{array}{l}
[\quad \Lambda[f_0 \mapsto \Lambda.\text{paxos}_{\text{fun}}(v),\ [\text{acceptors} = \langle \alpha_0, \alpha_1, \alpha_2, \ldots \rangle, \text{paxos}_{\text{fun}} = \ldots, \ldots]] \\
\quad\ \alpha_0[\varnothing,\ [\text{acceptors} = \langle\rangle, \text{paxos}_{\text{fun}} = \ldots, \ldots]], \\
\quad\ \alpha_1[\varnothing,\ [\text{acceptors} = \langle\rangle, \text{paxos}_{\text{fun}} = \ldots, \ldots]], \\
\quad\ \alpha_2[\varnothing,\ [\text{acceptors} = \langle\rangle, \text{paxos}_{\text{fun}} = \ldots, \ldots]], \\
\quad\ \ldots \\
]
\end{array}$$

We will next step through the various fields and methods of an agent, explain informally their purpose and whether they belong to the role of an agent as proposer or acceptor. In the next section we will discuss alternatives and what the implementation tells us about the expressiveness of $\mathrm{ASP}_{\mathrm{fun}}$.

The status of an acceptor needs to store the last accepted proposal – which is only a natural number – and the last accepted value – a number and a value. We define the agent fields 'lap' (last accepted proposition) and 'lav' (last accepted value) initializing them with defaults.

$$\begin{array}{lcl}
\text{lap} & = & 0 \\
\text{lav} & = & (0, \langle\rangle_{\text{val}})
\end{array}$$

For the rôle of an acceptor we need methods to manipulate these two memory fields. If an acceptor receives a new proposal n of the leader in phase 1 of the algorithm, it needs to record this in lap unless it is a proposal that is less than the last proposition. Therefore, each agent has the method choose_prop.

$$\text{choose\_prop} \quad = \quad \varsigma(x, n)\ \text{if}\ n \geq x.\text{lap}\ \text{then}\ (x.\text{lap} := n, \langle\rangle_{\text{obj}})\ \text{else}\ (\langle\rangle_{\text{obj}}, x)\ \text{end}$$

Similarly, each agent has a method acc_val that changes the lav field of the object if the value (i.e. its order number) is not less than the last proposition.

$$\text{acc\_val} \quad = \quad \varsigma(x,(n,v))\ \text{if}\ n \geq x.\text{lav}\ \text{then}\ (x.\text{lav} := (n,v), \langle\rangle_{\text{obj}})\ \text{else}\ (\langle\rangle_{\text{obj}}, x)\ \text{end}$$

In both methods we encode the acceptance of the proposition or value by updating the corresponding fields *and* by the position (left/positive,right/negative) in the result pair. This is due to a peculiarity of ASP_{fun}: immutable objects (see Section 3.3). We will see next how the two phases use the acceptor methods. To iteratively apply these methods over all acceptors we use the function 'map' as a generic function over lists of objects. Intuitively, it can be implemented as another method of agents as follows.

$$\text{map} \quad = \quad \varsigma(x,(f,l))\ \text{if}\ l = \langle\rangle\ \text{then}\langle\rangle\ \text{else}\ \text{hd}(l).f :: (x.\text{map}(f, \text{tl}\ l))\ \text{end}$$

However, this has to be seen as some sort of meta-program to be resolved in a preprocessing step – because we do not have polymorphism in ASP_{fun}. We use map here to improve the readability of the algorithm. It can be easily replaced by spelling out these maps in all three occurrences.

Now, phase 1 is implemented as a recursive method that given a list of agents A (the possible acceptors) calls the choose_prop method in each of those objects. It then dissects the responses into positive promises (using a function dissect defined below) and then checks whether a quorum (also defined below) of positives is reached, i.e. more than half the acceptors promise to go with n; otherwise it recurs.

$$\begin{array}{lcl}
\text{phase}_1 & = & \varsigma(x,(A,n)) \\
 & & \text{let}\ A_0 = x.\text{map}\ (\text{choose_prop}\ n)\ A \\
 & & \quad (A_p, A_n) = x.\text{dissect}\ A_0 \\
 & & \text{in}\ \text{if}\ x.\text{quorum}\ A_p\ \text{then}\ (n, A_p) \\
 & & \quad \text{else}\ x.\text{phase}_1(A_p @ A_n, n+1)\ \text{end}
\end{array}$$

Note, that the recursion operates with the concatenation $A_p @ A_n$ not with A_0 nor the original A. This is because objects that do promise become – due to the special semantics (immutable objects) of ASP_{fun}– replaced by new updated objects. Hence, we need to keep the ones that accept as they were and merge them with the ones that did not accept and remain the same. The following method dissects a list of pairs of objects into a pair of lists of accepting and rejecting objects,

$$\begin{array}{lcl}
\text{dissect} & = & \varsigma(x,l) \\
 & & \text{if}\ l = \langle\rangle_{\text{list}}\ \text{then}\ \langle\rangle_{\text{list}} \\
 & & \text{else}\ (\ \text{if}\ l = (o, \langle\rangle_{\text{obj}}) :: l'\ \text{then addleft}\ o\ (\text{dissect}\ l') \\
 & & \quad \text{else addright}\ (\text{hd}\ \ l)\ (\text{dissect}\ l')\ \text{end})\ \text{end})
\end{array}$$

where we assume two more basic list processing functions addright and addleft as follows.

$$\begin{array}{lcl}
\text{addleft}\ x\ (l,l') & = & (x :: l, l') \\
\text{addright}\ x\ (l,l') & = & (l, x :: l')
\end{array}$$

A quorum, i.e. a majority is a simple predicate using the length of the list of the initial set of acceptors contained in an agent (really useful only in leaders).

$$\text{quorum} \quad = \quad \varsigma(x, A)\ \text{length}\ A > (\text{length}\ x.\text{acceptors})/2$$

The second phase of the algorithm assumes a successful phase 1 for some n and then extracts the highest value accepted so far, otherwise v. It continues to call the accept method acc_val on all acceptors that have promised something in phase 1 and succeeds if there is a quorum of acceptors that have accepted.

$$\begin{array}{lcl}
\text{phase}_2 & = & \varsigma(x, (A, n, v)) \\
 & & \text{let}\ (m, A_0) = x.\text{phase}_1(A, n) \\
 & & \quad h = \text{highest_val}\ A_0 \\
 & & \quad v_h = \text{if}\ h = \langle\rangle_{\text{val}}\ \text{then}\ v\ \text{else}\ h\ \text{end} \\
 & & \quad A_1 = x.\text{map}\ (\text{acc_val}\ (m, v_h))\ A_0 \\
 & & \quad (A_p, A_n) = x.\text{dissect}\ A_1 \\
 & & \text{in}\ \text{if}\ x.\text{quorum}\ A_p\ \text{then}\ x.\text{success} := (A_p, v_h) \\
 & & \quad \text{else}\ x.\text{phase}_2(A_p @ A_n, m + 1, v_h)\ \text{end}
\end{array}$$

The method highest_val now selects from a list of objects the value that has the highest ordering number.

$$\begin{array}{lcl}
\text{highest_val} & = & \varsigma(x, l)\ \text{fst}(x.\text{hval}\ l\ (0, \langle\rangle_{\text{val}})) \\
\text{hval} & = & \varsigma(x, (l, v)) \\
 & & \text{if}\ l = \langle\rangle_{\text{list}}\ \text{then}\ v \\
 & & \text{else}\ (\ \text{let}\ l = o :: l' \\
 & & \qquad \text{in}\ (\text{if}\ \text{fst}(o.\text{lav}) > \text{fst}\ v\ \text{then}\ x.\text{hval}\ l'\ (o.\text{lav}) \\
 & & \qquad\quad \text{else}\ x.\text{hval}\ l'\ v\ \text{end}) \\
 & & \qquad \text{end})\ \text{end})
\end{array}$$

Finally the method $\text{paxos}_{\text{fun}}$ offers the algorithm as a method to a client object.

$$\text{paxos}_{\text{fun}} \quad = \quad \varsigma(x, v)\ (x.\text{phase}_2)(x.\text{acceptors}, 0, v)$$

3.3 Discussion

3.3.1 No livelocks with $\text{Paxos}_{\text{fun}}$

A typical livelock situation may occur in Paxos if two proposers exist [11, Section 2.4]: "proposer p completes phase 1 for a proposal number n_1. Another proposer q completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored. And so on." Both proposers never arrive at the end of their protocol successfully because in the meantime between the two phases the other one has given out a higher

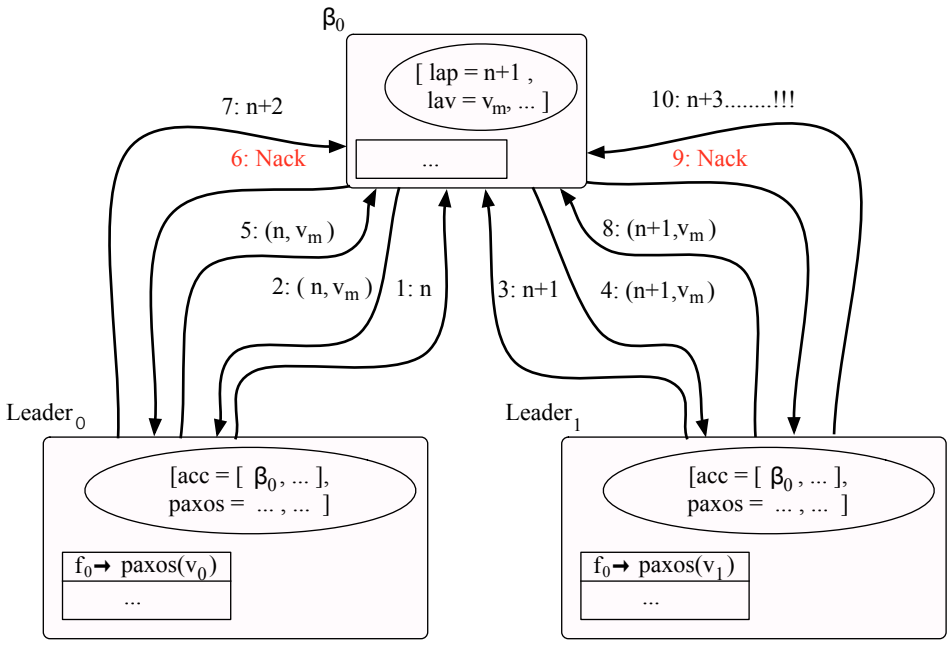

Figure 3.5: Possible livelock in Paxos.

proposition. The acceptors continuously upgrade their promises but they never accept a value. There is constant activity, one might even say progress, but the goal of the Paxos algorithm is never achieved; we call this a livelock. The livelock situation is depicted in Figure 3.5.

To overcome this situation, the only solution Lamport proposes [11] is to "...elect[ing] a single distinguished proposer."

In our language ASP_{fun}, the livelock cannot even occur simply because we are functional. To be more precise, this is due to the semantics of the update for active objects creating a copy of the original object with the fields updated. In our Paxos implementation in ASP_{fun}, whenever an acceptor saves a chosen proposition, this agent is copied, the copy stores the new lap or lav values corresponding to the proposition, and the copy replaces the original agent. The original agent in its original state is nevertheless preserved in case it features in some other context.

In the following it is this new object that represents – also from the point of view of the proposer – the (new) acceptor. Hence, if there is originally one community of acceptors to which two concurring proposers have access – as soon as they start proposing their conflicting numbers n – two new communities are created, each of them obeying one proposer. The livelock cannot occur. This phenomenon is illustrated in Figure 3.6

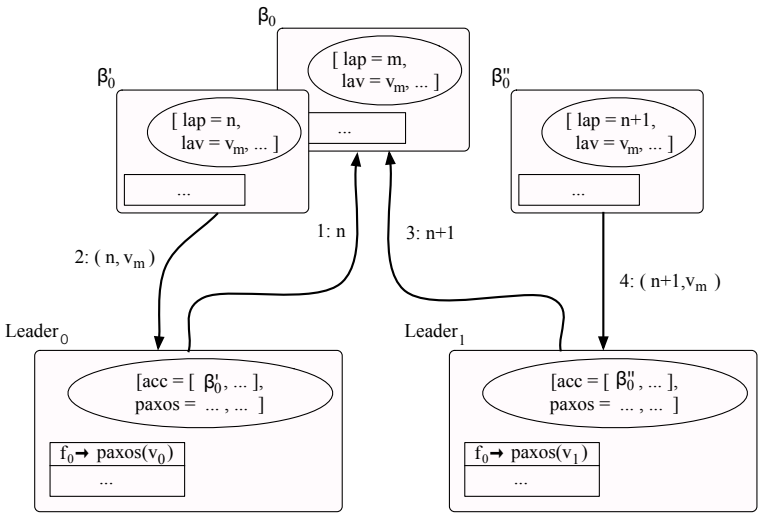

Figure 3.6: No livelock for Paxos in ASP_{fun}!

3.3.2 $\text{Paxos}_{\text{turbo}}$ in active objects with futures

In principle, the entire Paxos can be simply implemented with active objects and futures by a one line algorithm $\text{Paxos}_{\text{turbo}}$: proposer tells all its acceptors to accept value n. Since, the asynchronous communication in active objects with futures consists of request and response in one step – represented by a future – the loss of messages is not expressible. This is not specific to ASP_{fun}; it is true also for ASP and ProActive. Concerning the delay of messages, $\text{Paxos}_{\text{turbo}}$ would work in ASP_{fun}– because a reply is possible even for unevaluated futures – but in ASP and Proactive this would cause a so-called *wait by necessity*.

However, when implementing the above "$\text{Paxos}_{\text{turbo}}$" in ASP and ASP_{fun} there are differences. In a situation of concurring requests the ASP fun solution may deadlock through concurring requests, whereas the ASP_{fun} solution just produces two communities, as seen in the previous case of two proposers.

3.3.3 Modelling message delay and loss

Although the delay and loss of messages is not naturally implemented with our communication based on futures, we now explain how the Paxos implementation can be easily extended to simulate this. Since Paxos originates, presumably, from a Greek "part-time parliament" where parliament members could walk in and out of the parliament, message delay and loss is one decisive requirement to the algorithm. In fact, as we have seen in the previous section, if loss and delay are excluded, the algorithm becomes a trivial one line command.

To simulate the partiality of the parliament in our ASP_{fun} implementation

we simply add to the responses of the acceptors an additional randomness, based on a random variable $r \in \{0,1\}$. When an acceptor must choose or accept – according to the rules of the algorithm – he may now – in the extended version – decide to say nothing instead. The choice of which – to reply honestly or to pretend not being there – is based on the random r. We simply treat the pretence of absence in the same manner as a negative decision, thereby greatly simplifying this extension. From the point of view of the proposer, loss and delay must be treated in the same manner as negative responses. Concretely, we simply replace the two functions choose_prop and acc_val by the following adapted versions; otherwise the algorithm stays the same.

$$
\begin{array}{lcl}
\text{choose\_prop'} & = & \varsigma(x,n)\ \text{let } r = \text{rnd } \{0,1\} \\
 & & \quad \text{in if } (n \geq x.\text{lap}) \wedge r = 1 \\
 & & \quad\quad \text{then } (x.\text{lap} := n, \langle\rangle_{\text{obj}}) \\
 & & \quad\quad \text{else } (\langle\rangle_{\text{obj}}, x) \text{ end} \\
\text{acc\_val'} & = & \varsigma(x,(n,v))\ \text{let } r = \text{rnd } \{0,1\} \\
 & & \quad \text{in if } n \geq (x.\text{lav}) \wedge r = 1 \\
 & & \quad\quad \text{then } (x.\text{lav} := (n,v), \langle\rangle_{\text{obj}}) \\
 & & \quad\quad \text{else } (\langle\rangle_{\text{obj}}, x) \text{ end}
\end{array}
$$

3.3.4 Language based deadlock prevention.

When implementing the mutual exclusion problem – where each process claims exclusive control of a common resource – imperative ASP and functional ASP_{fun} show quite different features and different qualities of modelling. In this situation of concurring requests, the ASP solution may deadlock, whereas in ASP_{fun}, where no common data areas exist, each request just produces a new copy of the original object with the fields updated, as seen in the example of Section 3.3.1.

Bibliography

[1] A. Banerjee and D. A. Naumann. Stack-based Access Control for Secure Information Flow. *Journal of Functional Programming* **15**(2), 2003.

[2] G. Barthe and L. Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. *Formal Methods in Security Engineering, FMSE'04*, ACM-SIGSAC, 2004.

[3] G. Barthe and B. P. Serpette. Partial Evaluation and Non-inference for Object Calculi. Fuji International Symposium on Functional and Logic Programming, FLOPS'99. LNCS **1722**, Springer, 1999.

[4] D. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, **19**(5): 236–242, 1976.

[5] D. Denning and P. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, **20**(7): 504–8513, 1977.

[6] J. Goguen and J. Meseguer. *Security Policies and Security Models.* In Proceedings of SOSP'82, pages 11–22. IEEE Computer Society Press, 1982.

[7] L. Henrio, F. Kammüller. A Mechanized Model of the Theory of Objects. *9th IFIP Int. Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'07.* LNCS **4468**, Springer 2007.

[8] L. Henrio and F. Kammüller. Functional Active Objects: Typing and Formalisation. *Foundations of Coordination Languages and System Architectures, FOCLASA'09.* Satellite to ICALP'09. ENTCS, 2009.

[9] F. Kammller. Formalizing Non-Interference for A Small Bytecode-Language in Coq. *Formal Aspects of Computing*: **20**(3):259–275. Springer, 2008.

[10] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems***16** (2):133–169, 1998.

[11] L. Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column).* **32**(4 (Whole Number 121)): 51–58. December 2001.

[12] A. C. Myers and B. Liskov. Protecting Privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology. TOSEM*, **9**:410–442, 2000.

[13] A. Sabelfeld, H. Mantel. Securing Communication in a Concurrent Language. Static Analysis Symposium, SAS'02. LNCS **2477**, Springer, 2002.

[14] D. A. Naumann. *Verifying a Secure Information Flow Analyzer.* Theorem Proving in Higher Order Logics, TPHOLs'05, Oxford 2005. LNCS volume 3603, Springer, 2005.

[15] A. Sabelfeld and A. Myers. *Language-Based Information-Flow Security.* Selected Areas in Communications, **21**:5–19. IEEE 2003.

[16] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. *POPL'98.* ACM 1998.

[17] M. Strecker. *Formal Analysis of an Information Flow Type System for MicroJava (extended version).* Technical Report, Technische Universität München, July 2003.

[18] D. Volpano and G. Smith. A Type-Based Approach to Program Security. *TAPSOFT'97.* LNCS **1214**, Springer 1996.

[19] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, **4**(3): 167–187, 1996.